SIMULATING DISTRIBUTED DATABASE OPERATING SYSTEMS AND EVALUATING CONCURRENCY CONTROL PROTOCOL PERFORMANCE

By

Partha Dasgupta, Zvi Kedem and Yiu Leung[‡]

Technical Report #214
March 1986

SIMULATING DISTRIBUTED DATABASE OPERATING SYSTEMS AND EVALUATING CONCURRENCY CONTROL PROTOCOL PERFORMANCE

By

Partha Dasgupta, Zvi Kedem and Yiu Leung[‡]

Technical Report #214
March 1986

[*] Partially supported by NASA under contract number NAG-1-430, by NSF under grant number DCR-8416422 and by ONR under contract number N00014-85-K0046

[‡] Authors' Addresses

1) School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

(2) Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

3) 307 Middletown-Lincroft Road
Lincroft, NJ 07738

Simulating Distributed Database Operating Systems and Evaluating Concurrency Control Protocol Performance

Partha Dasgupta, Zvi M. Kedem, and Yiu K. Leung

1. Introduction

Research in the field of concurrency control for database systems has given rise to many techniques of ensuring consistency in multiuser database systems [BeGo80A, BeGo82B]. However claims of superiority of proposed protocols have mainly been supported by intuitive reasoning. Simulation is one of the methods that can be used to demonstrate efficiency and practicality of the mechanisms when analytical methods are not easily available.

The paper was motivated by our research on concurrency control and related issues. It has been repeatedly observed that concurrency control protocols for database systems are often untested in practical environments, unless of course, an actual database system uses some specific protocols. Sound analytic methods for comparisons of efficiency, throughput, and robustness are difficult to utilize because of inherent difficulties of dealing with the rather complex systems.

It is, however, of interest to be able to evaluate a variety of concurrency protocols and other database operating systems modules under identical conditions, to test which kinds are most suitable under different conditions and loads. We have thus designed a distributed database simulator, which if implemented would allow researchers to have one more tool in performance evaluation. For reasons explained in the paper we have decided at this point to test the concurrency control protocols, which were the focus of our interest in the special case of centralized environment only.

The simulator provides a concrete, and often the only practical way of judging the merits of different strategies under various conditions of operation. It can provide statistical insight into the different factors that affect performance, and the correlation of these factors with desirable features. It can thus be also used for fine-tuning existing protocols. Finally, it can be used as a verification tool and for enhancing our intuition about the issues involved.

We first describe very briefly some of the previous research dealing with database performance evaluation; it serves mainly to lead the interested reader to relevant literature. We then proceed to the description of our model of the distributed database operating system and what needs to be accounted for in a useful simulator. Later we go into substantial detail about the simulation and implementation techniques to provide the reader with information about the exact simulation environment, sufficient to judge the validity and the usefulness of the results obtained. Finally, we describe the results of simulating several concurrency protocols and present our interpretations.

2. Prior Results

Performance evaluation has been a topic of interest to database research community. Concurrency control protocols in particular, have been studied using both simulations and analytic techniques. We now very briefly describe some of the published research on database performance.

Lin and Nolte [LiNo83] compare 2-phase locking and timestamps. However they do not consider CPU processing delays in transactions. Their version of the 2-phase locking protocol uses exclusive locks for all data items that may be written as they do not allow lock upgrading. Thus the locking protocol tested probably has a poorer performance than locking protocols that employ lock conversion strategies.

Kiessling and Landherr simulated various locking strategies [KiLa83]. However these results are not complete and miss some details. They ignore CPU and I/O delays, and the transaction processing environment is thus not complete.

Ahuja, Browne and Silberschatz [AhBrSi84] simulates locking protocols in an attempt to find performance enhancement through optimal locking. They use post mortem traces to reconfigure locking requests and show that the performance could have been improved by optimal locking strategies. However these locking strategies are not realizable in practice, and though the paper shows how much better optimum locking is better than ad-hoc locking, it is not clear how this result could be used in real databases.

Lin [Li81] compares and evaluates the SDD-1 protocol [BeShRo80] and a dynamic timestamp protocol using a simulator. The simulator assumes constant delays for communication and I/O. We feel that the exclusion of processing delays may cause the results to differ from those expected in an actual system.

Sevcik [Se81] discusses some analytical approaches to performance predictions in various database system configuration. Thomasian [Th85] uses a queueing model using Markov chains to simulate locking protocols having predeclared locking, and compares the dependence of performance on granularities of locking.

Tay et. al. [TaGoSu85] have presented an elaborate study on performance of locking protocols in centralized databases. They use an analytical model to predict performance levels of locking protocols, and these predictions are verified through simulations. The study shows that the upper bound on throughput is dependent on blocking due to conflicts.

Agrawal and DeWitt [AgDe85] compare integrated concurrency control and recovery techniques in distributed database systems. Some of their results regarding restarts in optimistic concurrency control are similar to ours. The results of [TaGoSu85] and [AgDe85] are interesting, especially as the simulation results support the analytical studies. However the comparative merits and demerits of the various protocols studied is not very obvious. The performance studies present a host of parameters that may sometimes be hard to interpret.

3. Database Operating Systems and their Simulators

Database systems can be logically divided into two parts, namely the *database management system* and the *database operating system*. The database management system implements data modeling, defines a database query language, manages the storage and retrieval of data consistently with the model of data used by the database system and provides interfaces to the user (interactive) or interfaces to the query processing language (batch).

The database management system runs on top of the database operating system. The database operating system receives requests of actual data reads and writes from the database management system. The database operating system is responsible for the handling of *concurrency, commit, failure recovery*, and *I/O execution*. In this paper we will deal with modeling and simulating the database operating system [Gr79].

The model of a distributed system that we are using is composed of several database machines connected by a network. Each database machine has a transaction processing environment complete with a *scheduler* (managing concurrency and recovery), a *data manager* (handling I/O), and a *network server* handling database requests on behalf of remote transactions [Li79].

The simulator should implement as closely as possible the actual conditions and environment in a database system. This implies all kinds of processing delays, input-output delays and communication delays should be modeled and accounted for.

3.1. Distributed Environments and Concurrency Control Performance

The main focus of this paper is comparing the efficiency of various concurrency control protocols. We also have an interest in studying the efficiency and reliability of all other components of a distributed database system, such as recovery, distributed commit protocols, network routing protocols, fault tolerance handlers, etc. To better understand the issues and problems involved we have completed a design of a distributed database simulator.

As described in detail later, the distributed simulator has as its building block a *site simulator*. This single-site simulator is in effect a centralized database simulator with communication support. We have implemented a site simulator and used it as a centralized database simulator to obtain the results described in this paper. Our reasons for not simulating distributed concurrency control protocols at this point follow.

Simulating concurrency control protocols in a distributed environment needs an accurate representation of a host of parameters that reflect upon the performance of the concurrency control protocols used, but are not a part of the concurrency control protocols themselves. In other words the performance of the system is dependent on a large number of factors other than concurrency control protocols. For example some of these factors are: network delays, routing techniques, hardware reliability, distributed commit protocols and so on.

As is apparent, in distributed systems concurrency control is far from being the only factor limiting the database operating system performance, and is quite likely that it is not even the dominating one. The efficient integration of the chosen concurrency control method with the networking, commit, and other protocols is of paramount importance in distributed systems design. Thus it is difficult to correlate concurrency control protocol performance with the overall system performance in a distributed system.

The scope of the paper is to evaluate the performance of some concurrency control protocols. In order to keep the simulation parameters small, relevant, and easy to understand we decided to test a few concurrency control protocols isolated from the complexities of a distributed system. Thus we have chosen to simulate the concurrency control protocols in a multi-user centralized environment. This admittedly may limit the scope of the results but provides a deeper understanding of the protocols' properties.

We present in some detail the design of the distributed database simulator and show how we use the site simulator to obtain results on concurrency control performance.

3.2. Simulating Actual Database Operating Systems

In actual systems designed for handling high database traffic, the database operating system is part of the operating system kernel. When a transaction runs, it makes read and write requests to the transaction scheduler. Actually the transaction invokes a database system library with high level database manipulation calls. The library functions (or database management system) translates these calls into accesses to data items stored in the underlying permanent storage. We will not deal with the algorithms of the translation, but consider them to be a part of the transaction itself.

In database systems transactions are created interactively by users or run in some arbitrary mix using batch processing techniques. The system has very little control over the type and attributes of the transactions. These transactions interact with the database system by issuing read and write requests on the data stored in the system.

The requests to read and write data items are channeled to the *transaction scheduler*, which is the central component of the database operating system. (Henceforth, we will use the term *scheduler* to refer to this transaction scheduler.) Using some concurrency control protocol the scheduler decides to *allow*, *delay*, or *abort* the request (delayed requests are allowed or aborted at some later point). All allowed requests go to a data manager or disk drives that actually perform the I/O. We will simulate the transactions, the scheduler, and the data manager.

Looking at each transaction, the time spent in the system is composed of five types, namely the *CPU* time, *ready-wait* time, when the transaction is ready to be run but is waiting for the CPU, *blocked* time due to delaying of its requests by the scheduler, *I/O blocked* time due to waiting for access to the disk unit responsible for servicing the I/O request and the *I/O wait* time when the disk is actually performing the I/O. In distributed databases a significant amount of delay is introduced by the *communication* time needed for sending messages to remote sites, to access remote data items.

There are certain additional types of delays. Some amount of time is spent for communication, between the transaction and the scheduler and between the scheduler and the data manager. Also some amount of time is spent for *processing* the scheduler decisions.

In most actual database systems, designed for efficient database processing, the transaction to scheduler communication and the scheduler to data manager communication are not done via message passing. That is the scheduler and the data manager are not processes that receive messages from the the transaction processes. We assume that the scheduler and data manager are implemented as procedures in database operating system kernel. Without going into great detail of these actual systems, this method of implementation allows the scheduler and data managers to be invoked by the invoker without any message passing delays or overhead (they are either linked-in procedure calls or are called through traps), and allow the execution of these procedures to be multi-threaded. This is of course true only for local communications, that is when all of these components are located on the same machine. There are substantial communication delays when the transaction talks to a scheduler at another machine, in distributed databases, because this kind of communication has to be implemented as message passing over some kind of a network. The structure of an actual database system matching our model is shown in Fig. 1.

Some database systems implement the transaction to scheduler communication and the scheduler to data manager communication by means of message passing. Thus the scheduler and the data manager are processes that receive messages from the transaction processes. These implementations are somewhat less efficient. However the reason they are designed in such a manner is the limitations or constraints imposed upon the system by the underlying operating system that limit the implementation to follow this approach. We chose not to simulate these types of implementations.

We will simulate, closely, the above model of a database system. We will account for all the five types of time consumed by the transactions, as listed above. We will however *not* account for the communication delays for local invocations of the scheduler and data manager, as they are insignificant, and we consider them to be a part of the transaction's CPU processing (this is very realistic as there is no context switching involved in these calls, and the transactions thread of control executes the scheduler and data manager algorithms.)

3.3. Simulator Structure

The simulator is designed to run on the Unix environment using the Berkeley Unix 4.2. The system dependent facilities used by the design are relatively small and well structured to be applicable and portable to virtually any multiprocessing environment.

Each site of the distributed database is a Unix process. These processes communicate with each other to represent inter-site communication. The communication is handled at a low level by the Berkeley Unix 4.2 IPC system, which provides reliable interprocess message passing. The processes however do not communicate directly but talk to a special process called the *network simulator*. When site *A* (represented by process *A*) needs to send a message to site *B* (process *B*), *A* sends the message to the network simulator, which forwards it to *B*. Thus we have a star as our communication topology. This proves very useful for several reasons. The basic randomness of message ordering in a network system is provided by the Unix IPC mechanisms. The network simulator can simulate any desirable network topology by using appropriate distribution of forwarding delays, queueing, etc. The network simulator can also simulate network failures and partitioning (by not forwarding some or all messages) for testing recovery protocols. By randomly discarding messages, we can have an unreliable message passing scheme for testing fault tolerant protocols.

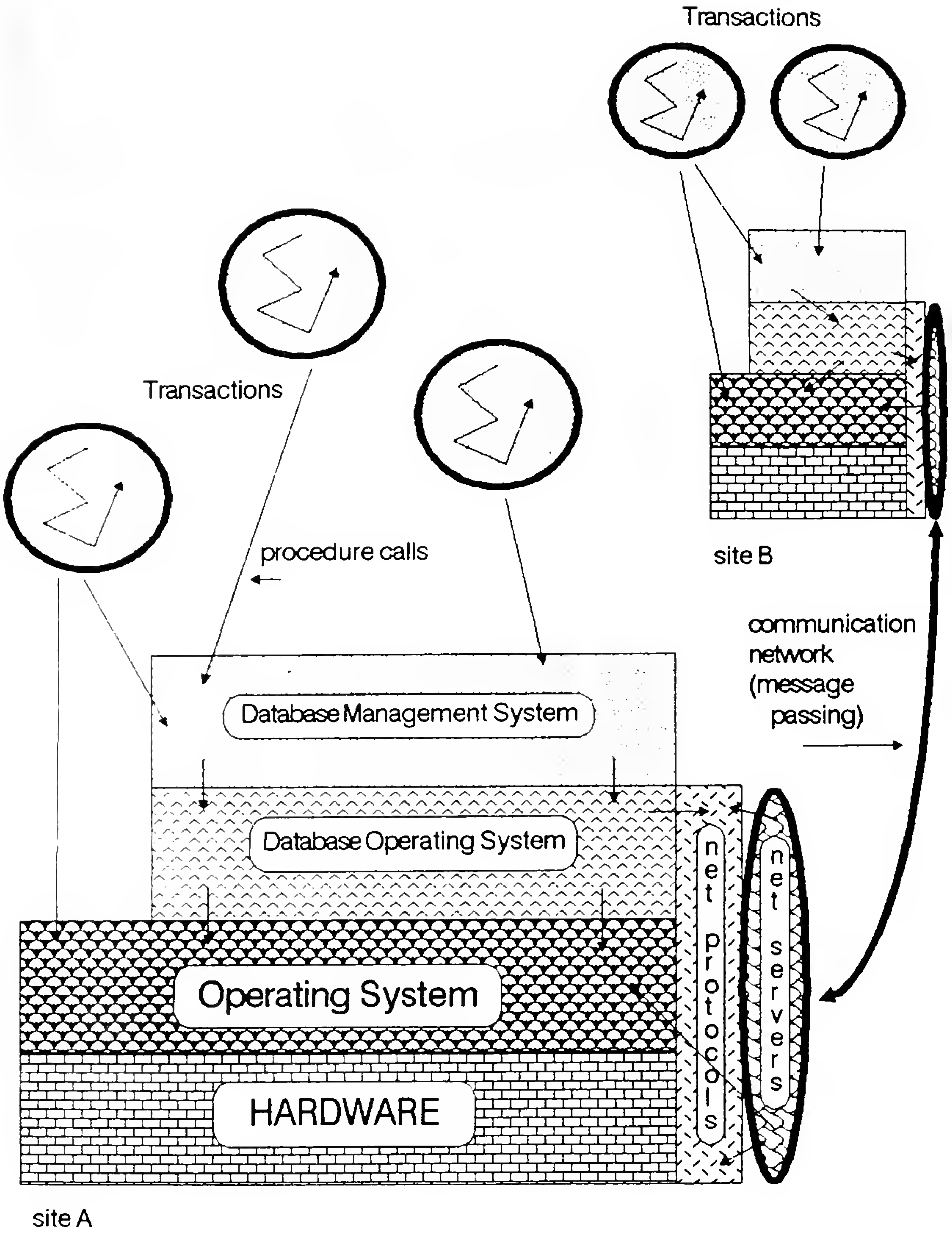


Fig. 1. Logical Structure of a Distributed Database.

The star topology might seem to have the problem of bottlenecks, if the network simulator is not fast enough, (or if it is not scheduled by Unix to run as often as needed). However this is easily handled in our simulator environment. The network simulator can be made to run arbitrarily faster than the site simulators, by slowing down the clocks in the site simulation processes. The logical configuration of the simulator is depicted in Fig. 2.

Each site simulator process consists of the following major data structures and algorithms. A set of transaction descriptors (described in section 3.4) keeps track of the simulated transactions. The process has a set of entry points (procedures) for the CPU handler, which implements a round robin CPU scheduling protocol, an I/O handler that has separate queue for the disks and handle disk I/O requests properly and the transaction scheduler that implements the concurrency control and the recovery (commit) protocols.

3.4. Simulator Implementation

The simulator is written in Pascal and currently runs on a VAX under Berkeley Unix 4.2. The interprocess communication routines that sends and receives messages from the network simulator are implemented as external stubs, written in C. Only these routines need to be replaced if the simulator is ported to a different environment. Each site of the database is simulated by a Unix process. This process will be composed of five modules: *simulator driver, transaction generator, CPU handler, communications handler, transaction manager,* and *I/O handler.*

The transaction generator generates a transaction every time it is invoked. It returns a *transaction descriptor* which has all the details about a transaction. The descriptor contains the sequence of reads and writes the transaction intends to do, the data it reads and writes. The transaction descriptor also contains the CPU time the transaction uses for each CPU burst (between two I/O requests) and the I/O time the disk unit takes to read or write each data item during each I/O operation.

The data accessed by the transaction may be local as well as remote. However when simulating the centralized database, as mentioned above, the accesses to local data only is generated.

The transaction descriptor is a very important data structure in the simulator. The information contained in the transaction descriptor ensures that if the same set of transactions are generated, the transactions will be simulated in an identical fashion over various runs of the simulator which uses different concurrency control protocols. This allows us to ensure near perfect uniformity over the test conditions for all the candidate protocols. The ranges of values for CPU and I/O delays, are chosen after some experimentation with system utilizations and are explained in in section 4.4 on simulation parameters.

The transaction generator can be programmed to generate a particular mix of transactions, and to control the pattern of accesses to data items (high traffic items, low traffic items, mainly read (or write) items, etc.) The transaction descriptor structure is shown in Fig. 3.

The driver controls the ticks of the simulation process. When a transaction needs CPU processing, the driver sends the transaction descriptor to the CPU handler. When it requests I/O, the driver submits the request to the transaction manager. If the request concerns a remote data item, then the transaction manager sends the request to the communications handler. The communications handler sends the requests to the network simulator. The communications handler also receives requests from the network controller, which it forwards to the local transaction manager. The pattern of accesses, and the local/remote ratios can be selected by the transaction generator. As we are discussing the centralized simulation results, the transaction generator does not generate remote accesses, and the communication handler and network simulator are not used.

The concurrency control protocol (scheduler) under test is plugged into the transaction manager. We replaced the scheduler protocol for different runs of the simulator to obtain results for different protocols. Achieving consistency and fairness for different runs of the simulator that use different protocols is discussed in section 4.1.

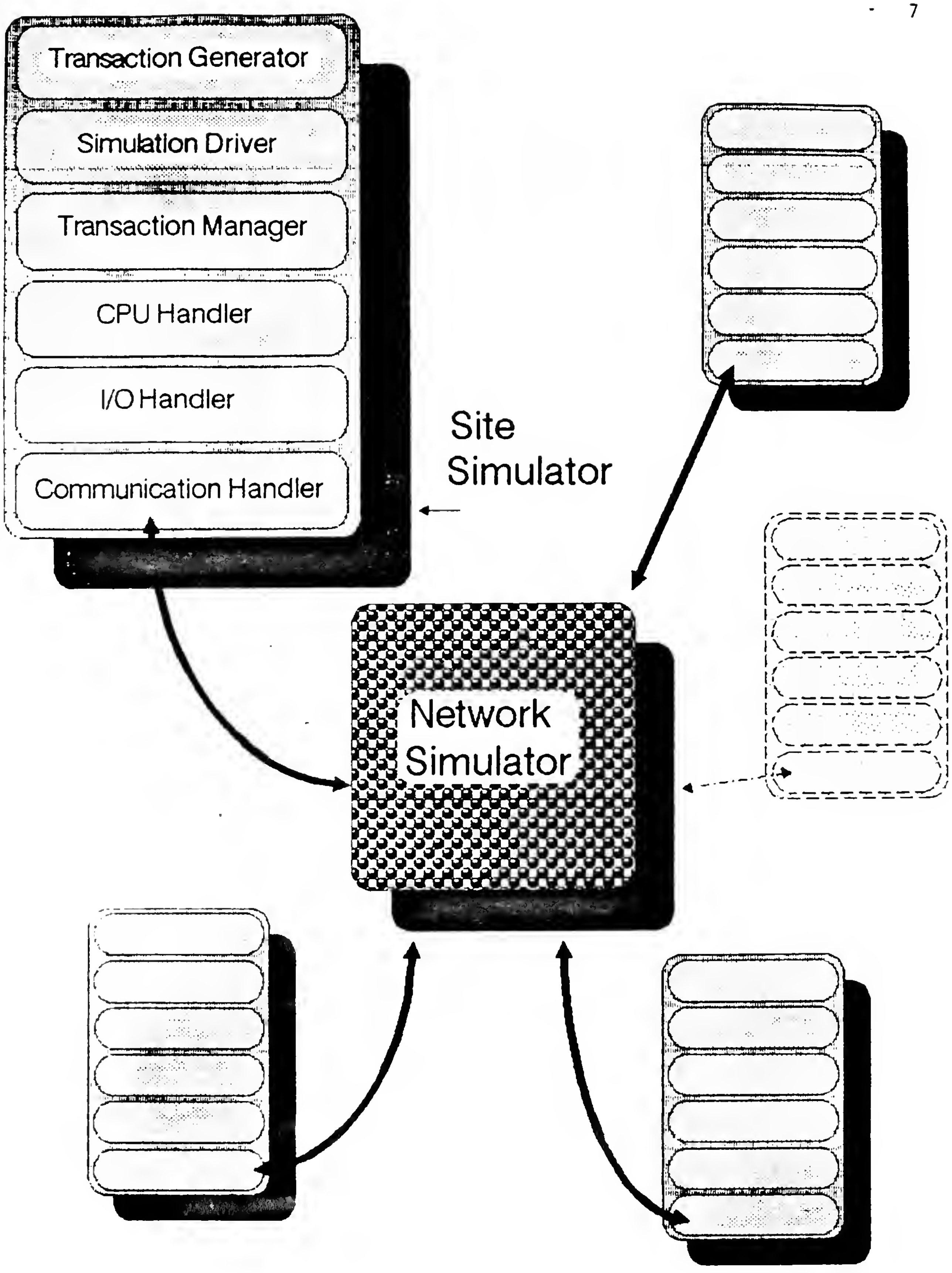


Fig. 2: The Distributed Database ~'mulator

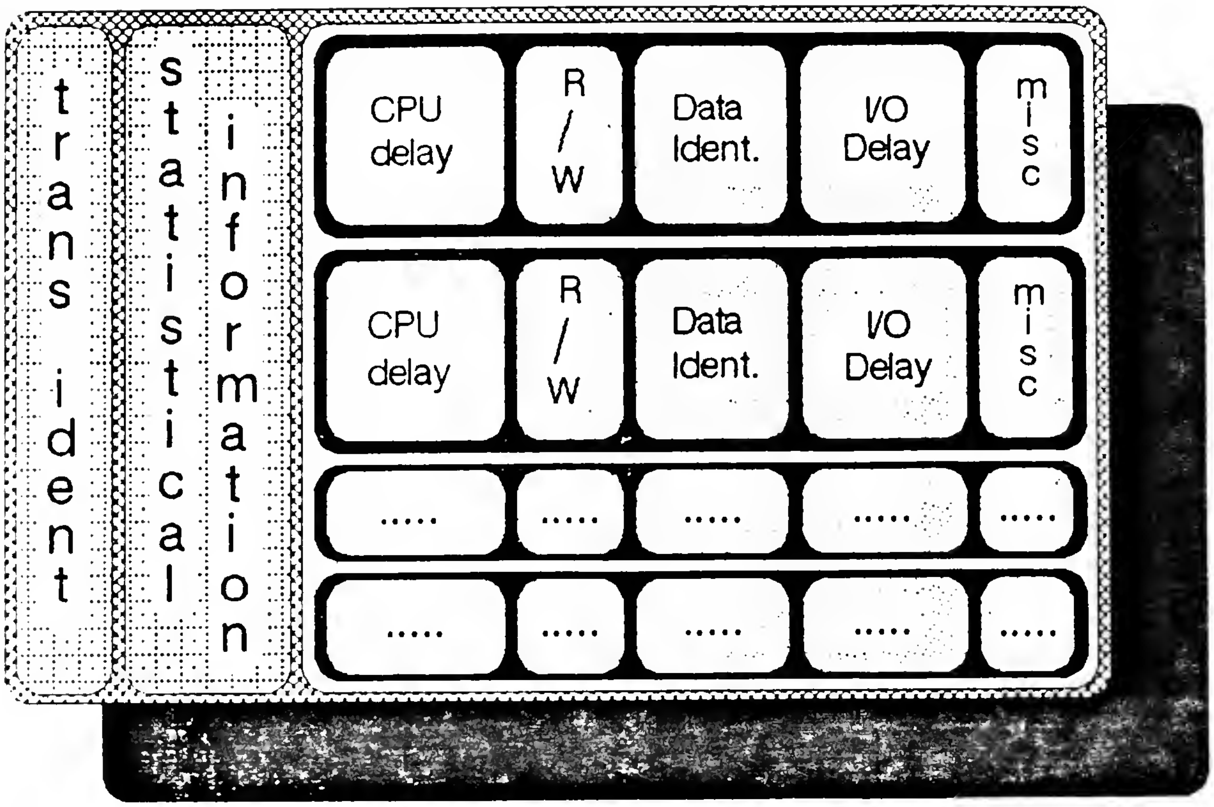


Fig. 3: The Structure of the Transaction Descriptor

The CPU handler maintains a run queue of transactions. Each transaction in the run queue consists of a pointer to its activity table, which marks its progress. The run queue is processed in a round robin fashion, with time slices alloted to each transaction. A transaction whose time slice runs out before its processing time is over, is sent back to the tail of the queue. However if the required processing is over and the transaction requests I/O, then the transaction is sent back to the driver, which forwards the request to the transaction manager.

The transaction manager receives a stream of requests from both local transactions (from the driver) or remote transaction (from the communications handler.) Each request is the invocation of the scheduler with the appropriate parameters. The remote requests from local transactions are forwarded to the communications handler. The concurrency control protocol (scheduler) is built into the transaction manager. The scheduler schedules each request as is consistent with the protocol involved. When the scheduler accepts a request, it sends it to the I/O handler. If the protocol is prone to deadlocks, the deadlock detector has to be built into the transaction manager.

The data manager simulates the actual I/O processing. It has one or more I/O queues depending upon the number of physical devices present at each site. The data manager looks at the head of the I/O queue and delays the request by a time period that is specified as the I/O time estimated in the transaction descriptor. When this time runs out, the data manager signals the transaction manager that I/O is complete for this particular transaction. The transaction manager sends back the acknowledgement to the driver (or the communications handler, if the request was remote). The driver returns the transaction to the CPU handler for additional processing, if needed. The logical structure of the site-simulator is shown in Fig. 4.

The driver controls the *timing* of the simulated environment. The driver gets hold of the program control at each simulated *clock tick*. At each tick, the driver first passes control to the CPU handler. The CPU handler reduces the remaining time on the current transaction by one. If this causes the timeslice to run out, or the CPU processing burst to end, the CPU handler removes this transaction from the run status to the appropriate status and schedules the next active transaction.

Then the control passes to the I/O handler. The I/O handler reduces the time remaining on the disk access wait record for all transactions at the head of each disk queue by one. If this action causes any disk I/O to complete, the transaction is returned to the ready state and the next transaction in the disk queue is scheduled for disk I/O.

The transaction manager then gets control of the program thread of execution. It checks to see if there are any pending requests and takes care of them as dictated by the concurrency control protocol. Then the driver increases the time by one tick and repeats the entire process.

4. Assumptions, Accuracy, and Measurements

An important area in any simulation study is the collection and interpretation of simulation data. Not only does the data have to reflect the real life situations closely but the simulator must accurately gather the data. In our simulator, the two things we have be to acutely concerned with, are gathering timing information and being absolutely fair over the various runs, using different protocols. In addition, as is true with all simulation studies, the parameters used must reasonably reflect real world situations. Also we have to be reasonably convinced that there were no bugs in the simulator that affect the implementations of the various protocols and the measurement routines. We describe the methods we used and the assumptions we made in order to achieve our goals.

4.1. Fairness

The successive runs of the simulator use different concurrency control protocols. We have to ensure that the transaction processing environments and the transactions that are run on the simulated systems are identical when the different protocols are used. The *transaction descriptor* described above in section 3.4 ensures this property.

As specified earlier, the transaction descriptor is a complete specification of the behavior of the simulated transactions. It contains information about the running time for each CPU burst, what the transaction reads (or writes) at the end of each CPU burst and the time the disk needs to access this data item. This completely describes the necessary data needed to duplicate the transaction. By using the pseudo-random number generator we ensure that the transaction generator produces the same set of transactions for all concurrency control algorithms.

4.2. Measuring Time

The two main and relevant measurements in our simulations are *throughput* and *dilation*. They are quite closely correlated. Throughput is the number of transactions processed by the system in a fixed amount of time, generally one run of the simulations. Dilation is the ratio of the actual running time of a transaction and the time the transaction would take to run in a serial mode.

The simulation driver actually keeps track of the simulated time. The time simulated by the simulator is measured as *ticks*. As described above, at each tick, several things happen. The CPU advances processing by one tick, the disk advances I/O time by one tick, and the transaction manager completes the computations for all pending requests. Note that there is a tacit assumption here that a step of the transaction manager *does not use the CPU, does not consume any time, and is essentially free*. This is not an accurate rendition of the actual database system, where the transaction manager does constitute a

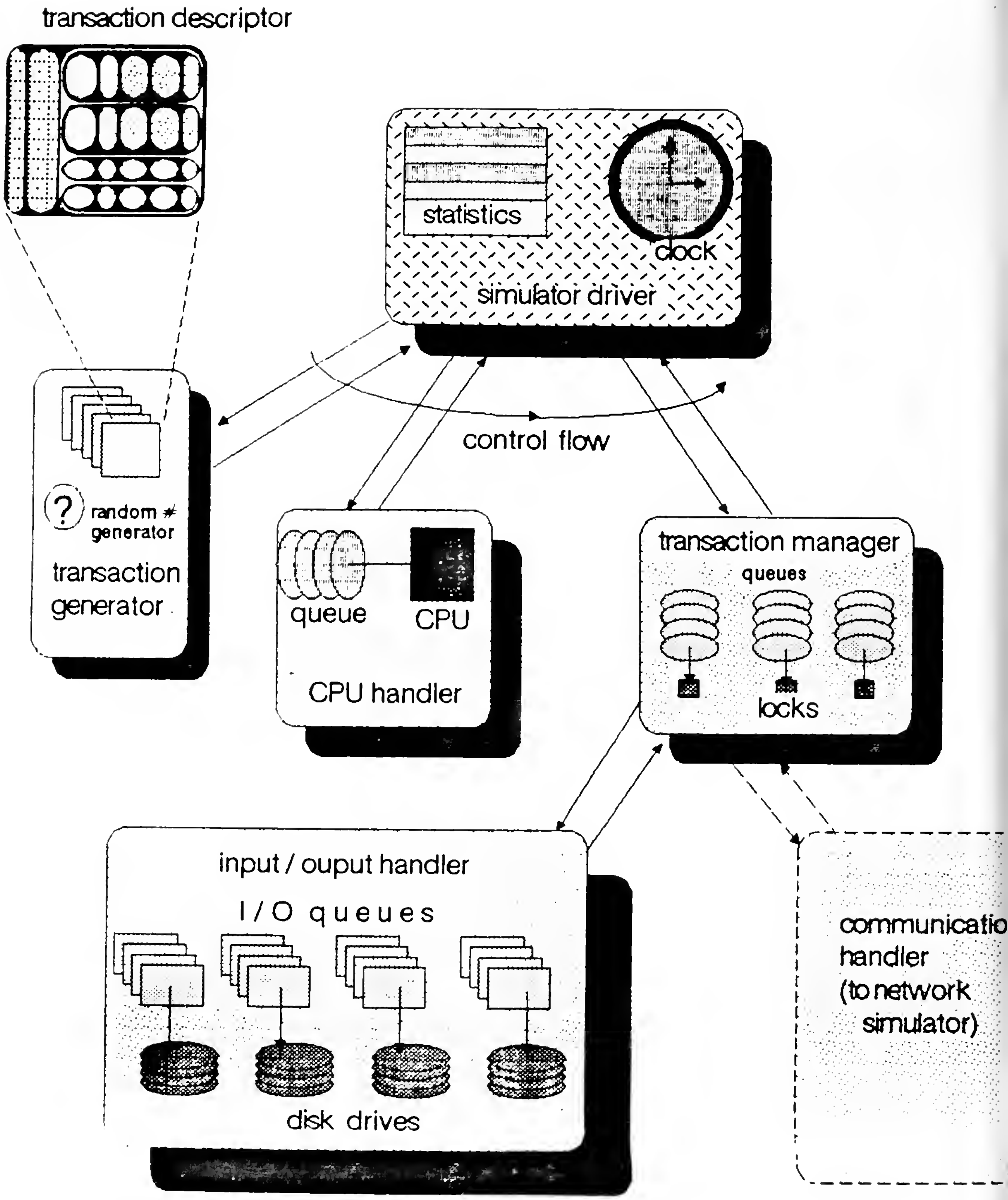


Fig. 4: The structure of the site simulator.

load on the system. The main reason behind this problem is that the transactions run on a simulated CPU while the transaction manager algorithms run on the actual CPU of the host system (a VAX in our case). The relationship between the VAX CPU and our simulated CPU is difficult to establish.

We feel we could have taken the overheads of the transaction manager into account by some clever and complex accounting process that stole parts of the simulated CPU time from the process according to measurements of the time taken to run each transaction manager procedure on the VAX CPU. We chose not to do this for the reason described below.

4.3. Ensuring Simulator Correctness

One of the major problems with simulation results, which have tainted the reputation of simulations is the difficulty of judging whether the program really does what it is supposed to do. Bugs in the simulator can produce totally meaningless results which still look reasonable. Thus the results may be used as correct, and the bug may never be suspected, let alone found.

In order to minimize the chances of this happening we consciously made the simulator as simple as possible without sacrificing accuracy to great extent. The simulator is simple, straightforward, and we hope elegant. Good readability of the implementation was enforced, which enabled us to have code implemented by one person, independently verified by another.

Thus simplicity disallowed us from choosing to account for overhead. This means that a protocol with a higher overhead may marginally perform better than it should. That is however not necessarily true. Please see the discussions in sections 4.4 and 7 about how our simulator showed that the CPU usage of the system was not very high. In such a situation, a higher overhead would simply soak up the extra wasted CPU time and not have any major effect on the throughput. Thus we believe this choice does not compromise accuracy in any significant way.

4.4. Simulation Parameters

The unit of time in the simulator is a *tick*. A tick is the smallest time interval the simulator can detect. All scheduling takes place at tick boundaries. As stated each transaction runs on the CPU for a certain amount of time, then requests I/O. This repeats for a certain number of times depending upon the number of I/O requests a transaction makes. The CPU time needed by a transaction between I/O steps varies between the parameters *MINCPU* and *MAXCPU*. The slice of CPU time that is allocated to a transaction each time it gets to use the CPU is defined by *RUNSLICE*. (We use a preemptive round robin CPU scheduler.)

The I/O delay (disk access time) needed for each I/O operation varies between the parameters *MAXIO* and *MINIO*. The number of I/O requests that a transaction makes in its lifetime varies between 1 and parameter *MAXWORK*. The number of *DATA* items in the database is controlled by the parameter *DATA*, and these data items are equally divided amongst *DEVICE* number of disk units.

Each data access of the transaction is either a read or write. The ratio of all reads on the database and all database accesses is *RWRATIO*. Thus the probability that any access is a read is also *RWRATIO*. A transaction need not write the data items it reads, but in practice, the data items that are written by a transaction is quite dependent upon the data items read by the transaction. In order to model this, we programmed the transaction generator to ensure that *RWSAME* of the writes issued by a transaction are on data items it has already read.

The load in the system is varied by the number of transactions that can run simultaneously. This is controlled by the parameter *LOAD*. Setting *LOAD* to 1 runs the transactions one by one, and this is equivalent to a *serial scheduler*. For protocols that are prone to deadlocks, a deadlock detector is run every *CHKDEAD* ticks. The following are the values (or ranges of values) of these constants:

RUNSLICE	5		
MAXCPU	10	*MINCPU*	3
MAXIO	50	*MINIO*	10
DEVICE	4	*DATA*	100 to 200
RWRATIO	50% to 80%	*RWSAME*	80%
CHKDEAD	200		
LOAD	1 to 20	*MAXWORK*	10

The values of CPU and I/O delays, and number of I/O devices are chosen to balance the CPU vs. I/O loads in the system to realistic levels. This was done by checking the CPU and I/O device utilizations for various values of *MAXIO*, *MINIO*, *MAXCPU*, *MINCPU* and *DEVICE*. The above values were chosen because they reflect, intuitively, the amount of time that computer programs take to perform I/O and processing between I/O requests, and also because they lead to CPU and I/O device utilizations ranging between 60% to 80% in most cases. Thus the system was neither heavily CPU bound nor heavily I/O bound. The values of *DATA* are varied to change the amount of conflict in the system.

5. The Protocols Simulated

The simulator has been used to simulate four protocols in a centralized environment. The protocols are named *NO*, *2PL*, *TS* and *5C*. They are described below.

5.1. The Protocol "*NO*"

This protocol does not implement any concurrency control protocol. It allows transactions to freely read and write data items, the data accesses being interleaved in any fashion, depending on chance. *NO* produces inconsistent executions, but defines the upper bound of concurrency in a database system. Though *NO* is not an useful protocol it is included in the performance tests to serve as a yardstick. It shows the performance of a system without concurrency controls, and illustrates the reduction in concurrency when other protocols are used.

When a transaction issues a read request on a data item x, the transaction manager always accepts the request. The transaction manager forwards the request to the I/O handler which actually performs the reading of x. When the transaction issues a write on a data item x, the transaction manager ignores the request, and immediately returns control to the transaction (that is, returns it to the CPU handler). After the transaction completes processing, all the writes are performed. This is done to simulate the commit mechanism used by all database protocols.

The following is a summary of the algorithm used by the transaction scheduler for this protocol:

```
Request      Action

read(x)
             send read request to I/O handler

write(x)
             record the write request and return to caller

end-of-transaction
             send all write requests to I/O handler
```

5.2. The Protocol "*2PL*"

This is a variant of the 2-phase locking protocol [EsGrLoTr76], that uses shared and exclusive locks [BeGo80]. When a transaction issues a read request on x, the transaction manager obtains a shared lock on x, and then the I/O handler reads x.

When the transaction issues a write request on x, the transaction manager checks to see if the transaction holds a lock on x. If it does not then it obtains an exclusive lock on x. Otherwise the shared lock already held is upgraded to an exclusive lock. Then control returns to the transaction and the write is not performed. No writes are performed as long as the transaction is running. After the transaction completes execution, all the writes are performed by the I/O handler (in response to requests from the transaction manager).

Deadlocks can occur and the deadlock detector is used occasionally. The aborts in *2PL* are due to breaking of deadlocks. The following code defines the scheduler algorithms for this protocol:

```
Request          Action

read(x)
          if (transaction does not have a lock on x) then
                    get shared lock on x;
          send read to I/O handler;

write(x)
          if (transaction has a shared lock on x) then
                    upgrade to exclusive lock
          else
                    get exclusive lock on x;


end-of-transaction
          for (all x exclusively locked by transaction) do
                    send write request to I/O handler;
```

5.3. The Protocol "*TS*"

This is the standard timestamp protocol [BeGo80A, BeGo80B]. Read requests are validated (timestamps of transaction and data item checked) and then read if validation succeeds. The write requests are ignored. After the transaction completes execution, the write requests obtained so far are validated and writing done (if validation succeeds).

Aborts in *TS* are due to validation failures. Cyclic restarts are quite common when load and conflict rates are high, as can be seen in the simulation results. The following shows the algorithm used by the scheduler:

```
Request        Action

read(x)
        if (timestamp conflict on x [1]) then
                abort the transaction
        else
                send read to I/O handler

write(x)
        record the write request and return to caller

end-of-transaction
        for (all x written by transaction) do
                if (timestamp conflict on x [2]) then
                        abort transaction
                else
                        send write to I/O handler
```

5.4. The Protocol "*5C*"

5C is an implementation of the Five Color protocol as described in [Da84, DaKe83, DaKe86]. The *5C* protocol is a locking protocol that uses a non-2-phase locking strategy on general databases, using information about the read and write sets of a transaction. As the entire activity schedule of a transaction is available as soon as the transaction arrives, the transaction manager can obtain the read and write set information from the transaction descriptor. The protocol is somewhat complex, and is not presented in detail here. One of the reasons for attempting this simulation study was to compare the *5C* protocol developed by two of the authors with the *2PL* protocol, and thus we include the results of the *5C* protocol in this paper.

Initially all the required locks are obtained, and the sets Before and After are built. If the transaction passes validation, the lock inheritance processing is performed. All the data items in the readset are read (the requests are forwarded, one by one, to the I/O handler). The transaction commences execution. During the execution phase, I/O requests are ignored, as I/O is done to/from local storage. After the transaction completes execution, the locks on the writeset are upgraded, and the data items written. Finally all locks are released.

Deadlocks can occur, but are expected to be rare, due to a deadlock avoidance scheme. Those deadlocks that still occur are detected by a deadlock detector that runs occasionally. Transaction aborts take place due to validation failures and when deadlocks are broken.

6. Simulation Results

We present the results from 80 runs of the simulator using the above 4 protocols. We divide the runs into four parameter set classes, each class consisting of five runs on each protocol.

Each run of the simulator generates a large number of transactions covering all types and mixes of transactions. Our experiments with varying the length of the run, and varying the seed for the pseudo random generator showed that for any seed value, running the simulator for 100,000 ticks produces the same results, showing that the simulator reaches steady state. The results in this paper uses runs lasting

[1] Timestamp conflict on read: w-timestamp of data item is higher than timestamp of transaction

[2] Timestamp conflict on write: w-timestamp or r-timestamp of data item is higher than timestamp of transaction

300,000 ticks. (This takes about 3 hours of CPU time on a VAX-11/750.)

The parameters that are varied are the number of data items in the database (*DATA*), the percentage of reads versus writes (*RWRATIO*) and the number of concurrent transactions on the system (*LOAD*). For each parameter set, we keep the *DATA* and *RWRATIO* constant, and vary the *LOAD* to obtain a graph (one for each protocol) showing the performance versus the load on the system. Performance is measured in terms of the throughput (total number of transactions processed), dilation (time expansion per transaction) and the number of restarts.

For each parameter set the *LOAD* parameter is varied from 1 to 20. A *LOAD* of 1 reduces any protocol to the serial protocol, and all of them should have identical performance. This is apparent as all out graphs intersect at *LOAD* = 1. A load of 10 can be construed to be "medium" load, while a load of 20 is high. Thus we are able to show the changes in comparative performance over varying system loads. Also the amount of conflicts in the system is controlled by *DATA* and *RWRATIO*. If the number of data items is small, and the transactions access about the same number of data items, then several transactions may access the same data item concurrently. But concurrent access does not cause conflicts if the accesses are read accesses. Decreasing *RWRATIO* causes an increase in the number of writes, and increases the amount of conflicts. The following subsections shows the results obtained in a database with varying degrees of loads and conflicts.

6.1. Parameter Set 1

The results for parameter set 1 is shown in Fig. 5. The *DATA* and *RWRATIO* is kept constant over the runs in this parameter set. The value for *DATA* is 100 and the value for *RWRATIO* is 80%. Thus we have a database having 100 items, transactions are reading 80% of the time, and the number of data items each transaction accesses varies from 1 to 10 (*MAXWORK*). Depending upon the load, this situation simulates a database having a low to medium amount of conflicts.

The load on the system is varied from 1 to 20. As is apparent from Fig. 5., all the protocols perform well when the load is 5. In fact they are hardly discernable from the *NO* protocol, showing that the effect of concurrency control is not significant, irrespective of the protocol used at low-load, low-conflict situations.

As the load is increased so is the conflict level. The *2PL* and *5C* protocol throughputs rise, the *5C* being somewhat better than *2PL* at high loads. The throughput of the *TS* protocol actually decreases as the load goes up, showing the effects of too many restarts. The graph showing the number of aborts clarify this observation. Thus this implementation of the timestamp protocol is unable to handle high load and conflict situations.

The results of the other parameter sets are similar, except for some interesting cases.

6.2. Parameter Set 2

In this set, the *RWRATIO* is decreased to 50% making the conflict rate quite high. The results are plotted in Fig. 6.

The point of interest in this set is the performance of the *TS* protocol. This protocol actually thrashes and hardly allows any transactions to complete. Most transactions get aborted when they try to write, and they keep being restarted cyclically. The abort rate is very high.

In fact the cyclic restart causes so many aborts that the throughput of the *TS* protocol with concurrent transactions is about 20% of the throughput of the serial scheduler. Thus this set shows that the timestamp protocol is quite impractical, in the form tested, for high conflict situations. However, this rate of conflicts is not expected in most actual databases, and this can be treated as a particularly hostile environment for concurrency control protocols. (Please see the discussion of timestamp protocols in section 7.)

The *2PL* protocol also shows some unexpected behavior. The protocol performs well up to a load of 10, and then the throughput drops sharply at 15. At 20 the throughput is still lower than the the

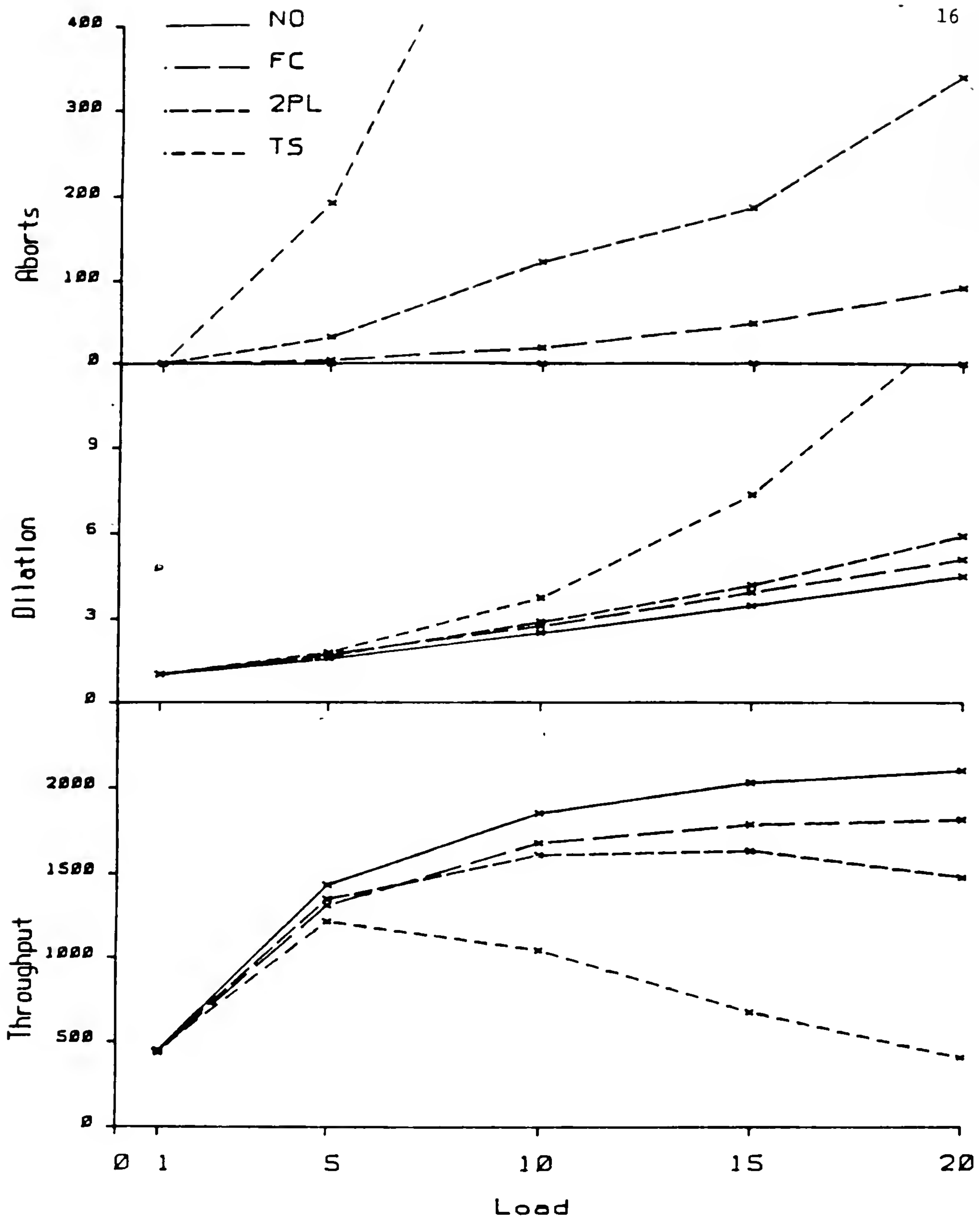


Fig 5 : Simulation Results for:
Number of Data Items: 100.
Percentage of Reads: 80%

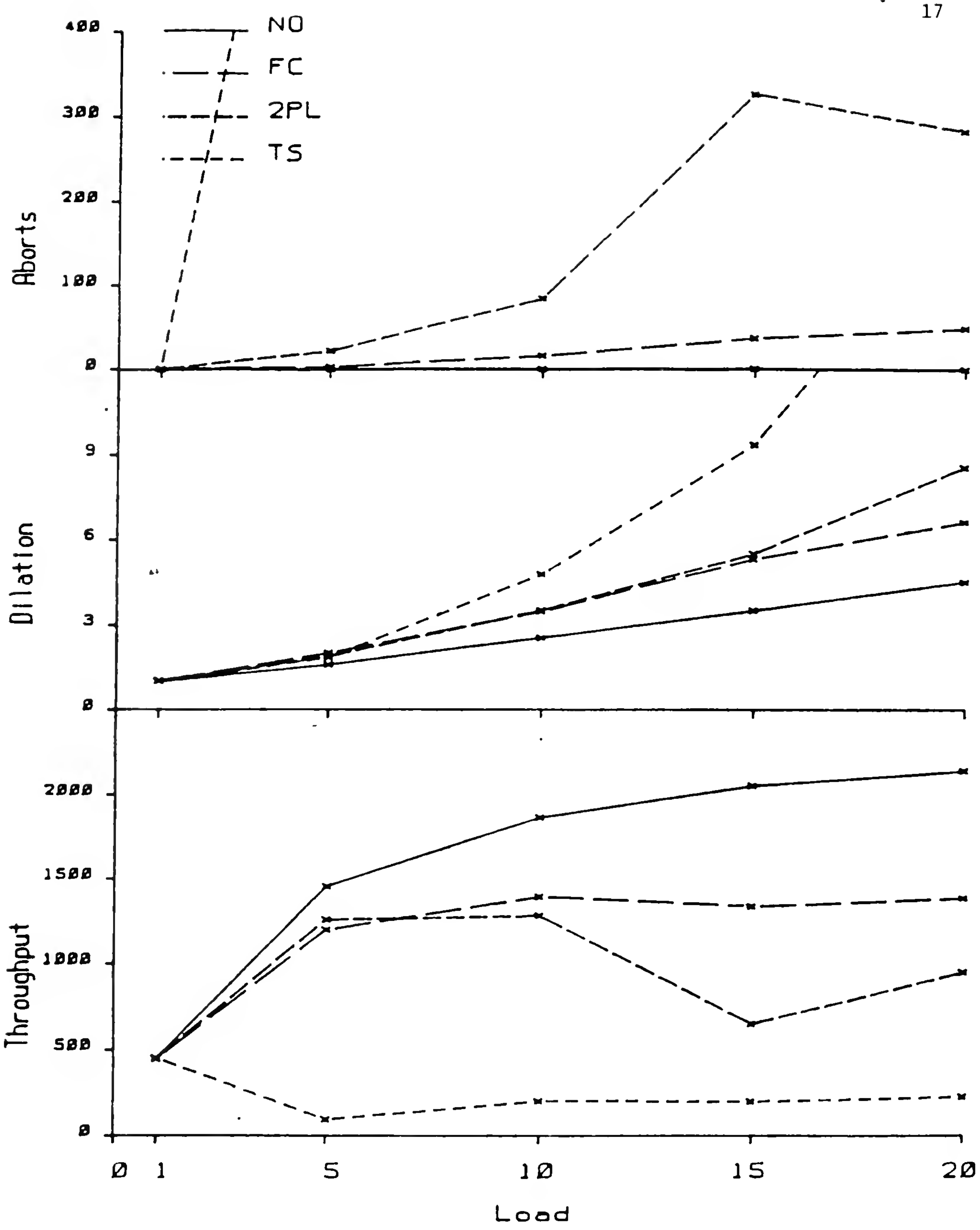


Fig 6 : Simulation Results for:
Number of Data Items: 1ØØ.
Percentage of Reads: 5Ø%

throughput at 10. Some trace analysis reveals that the *2PL* protocol is prone to thrashing at high loads too. Too many deadlocks contribute to a high rate of aborts causing the throughput to suffer. As the load is increased, some transactions actually manage to complete due to the high concurrency and the number of aborts also drops. There is no straightforward explanation to this behavior, except that thrashing cannot be modeled logically, and the results when thrashing is present are often counter-intuitive.

The *5C* protocol does quite well in the high conflict situation. Though the throughput does not increase with load, the number of aborts is contained, and the throughput is actually limited by the concurrency controller holding locks and preventing the transactions to run concurrently, which is necessary to ensure serializability.

6.3. Parameter Set 3

This parameter set uses a *DATA* = 200 and *RWRATIO* = 80%. The results are plotted in Fig. 7.

A larger value of *DATA* lowers the conflict rate and the database processes mainly reads, keeping conflicts low. This plot is a magnification of the low load, low conflict performance curves for parameter set 2.

As is obvious, the *TS* protocol does quite well till the load is 10. *5C* and *2PL* work nearly as good as *NO*, because the transactions hardly conflict. The *TS* throughput starts dropping as the load (and hence conflict rates) start edging up, and at a load of 20, the *TS* protocol again becomes inefficient.

The interesting part of this set is that it clearly shows that at low conflict rates, the *2PL* protocol is marginally better that the *5C* protocol. The *5C* protocol, intuitively, should work better than the *2PL* protocol for all cases. In fact *5C* is decidedly better that *2PL* whenever the load and conflict rates are medium to high. However, surprisingly, it does perform worse than *2PL* at extremely low loads. At first this seemed counter-intuitive, however, careful study of the scheduling queues and utilizations produced an explanation of this behavior. This is explained in section 7.

6.4. Parameter Set 4

Parameter set 4 uses the values *DATA* = 200 and *RWRATIO* = 50%. This simply is a enhanced conflict version of parameter set 3. *2PL* and *5C* are almost identical over the range of loads. They both trail the *NO* protocol by about 15% to 20% at loads of 25 to 20. The results are plotted in Fig. 8.

The *TS* protocol exhibits a classic case of thrashing: with the throughput falling, rising and falling again with no apparent correlation with anything. But the throughput nearly always is lower than the throughput of the serial scheduler. The *TS* protocol fares poorly whenever the number of writes is high.

7. Discussion and Conclusions

In this paper we present a simple and somewhat elegant design of a distributed database simulator that would prove useful in determining the performance of concurrency control, recovery, commit and network protocols for a distributed database connected by a networking system that is less than perfect. Each site of the database is a site simulator, which when run in a standalone mode, simulates, accurately a centralized database.

In order to test three concurrency control protocols, without considering the effects of the performance and reliability issues of network and commit protocols, we simulated the protocols in a centralized environment. Our simulator created a realistic environment and accounted for almost all types of delays and processing involved in database systems, that affect the performance of the system.

In our tests the parameters chosen yielded CPU and disk utilizations varying between 60% and 80%. In general, the CPU utilizations were lower than the disk utilizations, supporting the general belief that database systems are I/O intensive. It should be noted we used 4 disks against one CPU, in order to balance the load on the disks against the CPU load.

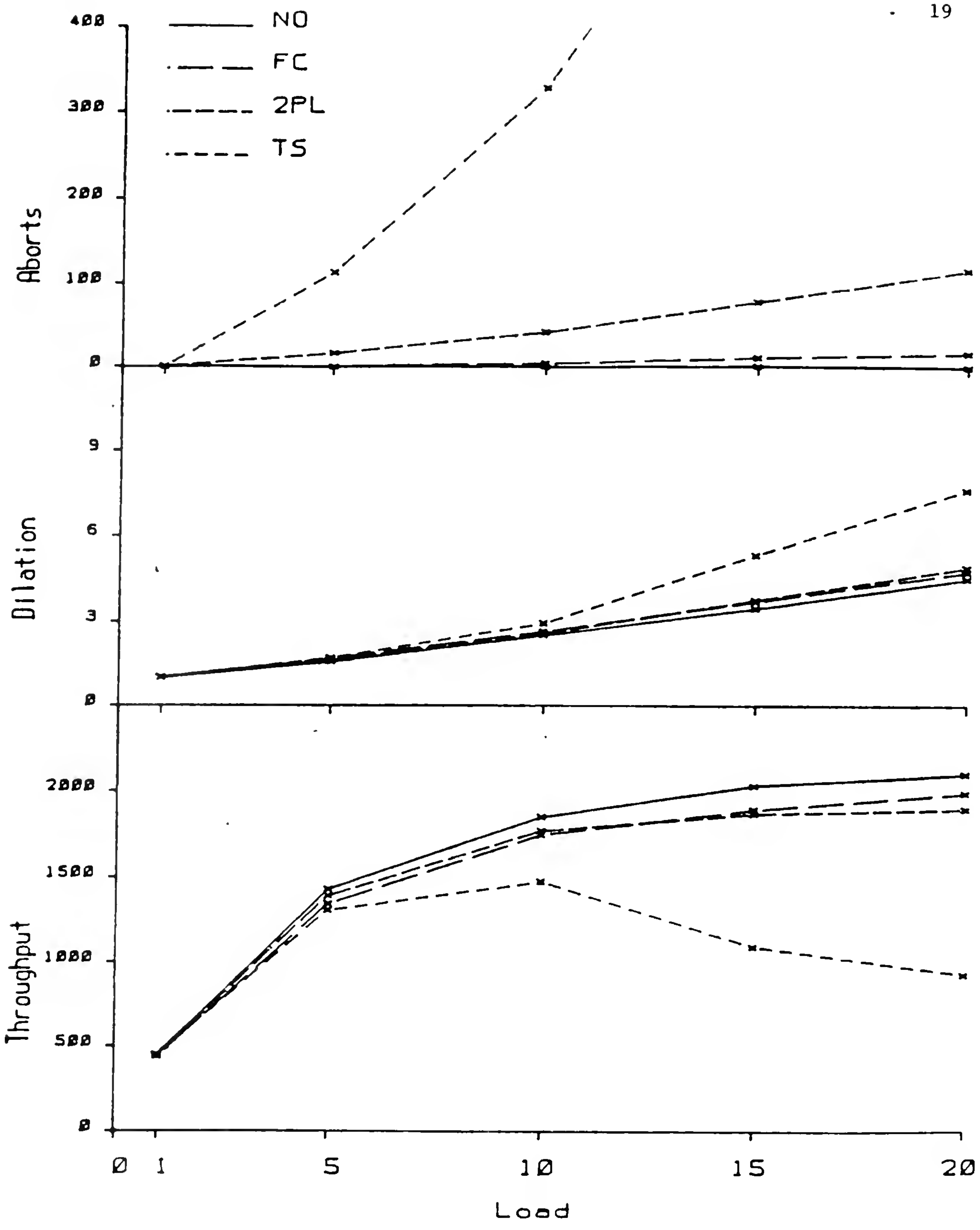


Fig 7 : Simulation Results for:
Number of Data Items: 200.
Percentage of Reads: 80%

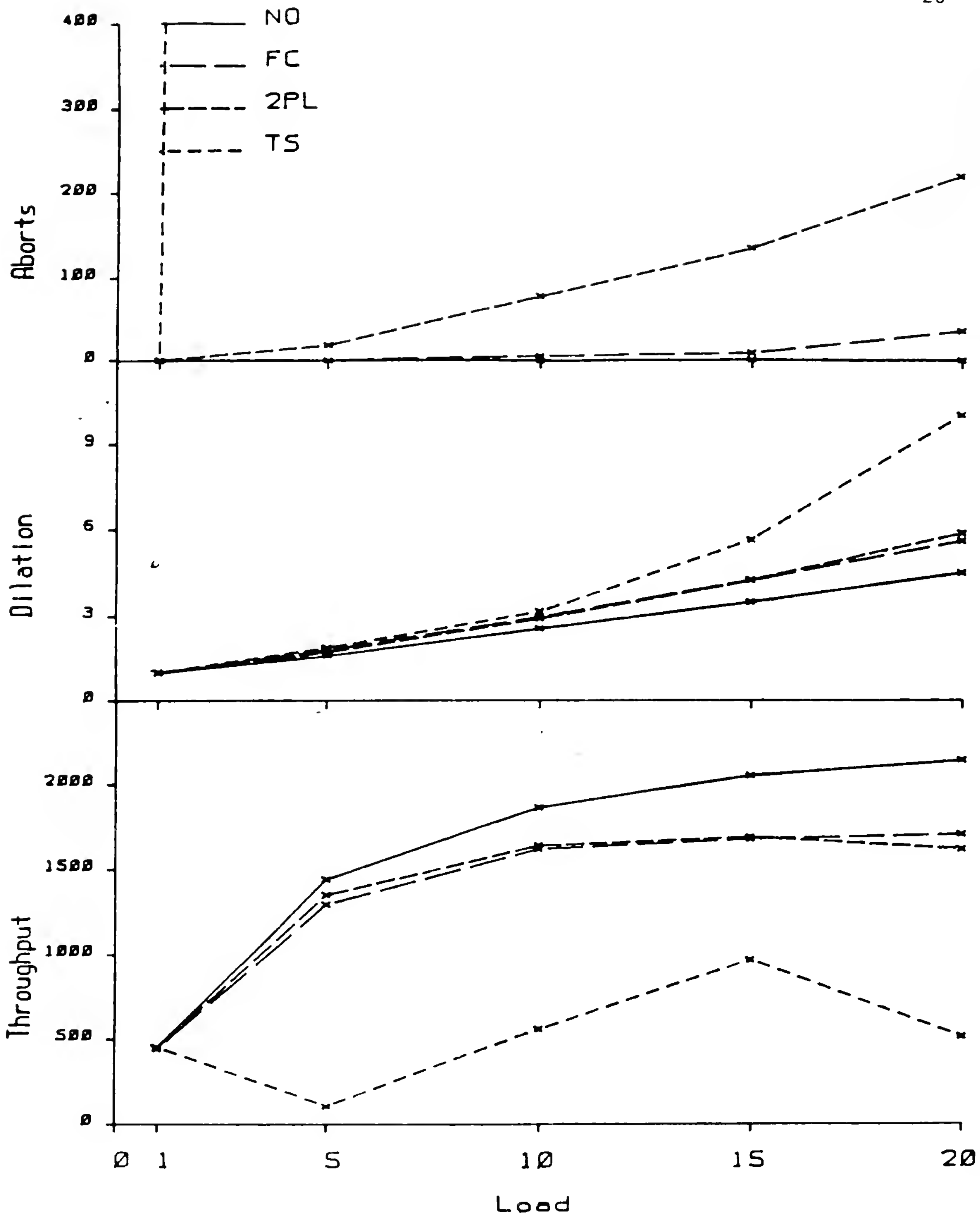


Fig 8 : Simulation Results for:
Number of Data Items: 200.
Percentage of Reads: 50%

The low CPU utilizations let us argue that some of the processing overheads that we chose to ignore (section 4.3) does not affect the results, because if we account for the overheads, they would simply increase the CPU utilization, by soaking up wasted CPU cycles in most cases, and not have any significant effect on the throughput.

The observed results were throughput, dilation and aborts. Dilation measures the *slowness* of the system from an interactive users point of view. This is directly proportional to the load when the no concurrency control is used. (This is apparent in the plots.) Higher dilation occurs when using concurrency control protocols because of waiting due to locking, or because of restarts. (The dilation of restarted transactions are computed with the total time taken to complete the transaction). The dilation of the locking protocols were slightly higher than *NO* (expected), but the dilation of the *TS* protocol was quite high due to cyclic restarts. The dilation figure for the *TS* protocol often is not relevant, in high abort situations, as the dilation is computed only for completed transactions, and very few transactions actually completed in some cases with the *TS* protocol.

The *NO* protocol, which does not use any concurrency control, is used as a yardstick for the upper bound of performance of any database system. Using any form of concurrency control is expected to lower the performance significantly with respect to the *NO* protocol. Comparing the locking protocols against the *NO* protocol yielded our first surprise.

At low loads and low conflict rates the locking protocols were just as good as the *NO* protocol. This is however expected, as there are hardly any conflicts, and the concurrency control protocols do not hamper any concurrency. As the conflict rates were increased, the performance of the concurrency control protocols fell somewhat below the *NO* protocol. However in a realistic environment of moderate conflicts and moderate loads (see *LOAD* = 10, in Fig 5.) the *2PL* protocol throughput was only about 12% lower than *NO*. At higher conflict situations, the performance of *2PL* was about 15% to at most 20% lower than *NO*.

This raises an interesting question: How much better can concurrency control protocols be, than 2-phase locking? Definitely any concurrency control protocol has to be significantly poorer that *NO*, especially in moderate to high conflict situations. We show that in extreme high loads, *5C* is better than *2PL*. But in some real situations, conflicts are expected not to be quite high. Also considering that the gap between *2PL* and *NO* is about 20% in medium to high load (medium conflict) situations, a significantly better protocol cannot be expected to have a significant performance increase in this load/conflict situation. An increase in performance by 5% or at most 10% seem to be realizable, but then the question turns to whether it is a worthwhile venture to hunt for better protocols given that the payoff is so limited.

Of course in special purpose applications where conflicts are expected to be high, we can get a performance payoff by using something better than *2PL* (*5C* performs about 20% to 40% better in the bad case depicted in Fig. 6).

The timestamp protocol performance was quite instructive. In a database that is not mainly read-only, poor performance is expected with the basic timestamp protocol. However we did not expect the results to be so drastic. Modifications to the timestamp protocol such as multiversions [BeGo82A] and intention locks for writes in general will likely make it more usable. Our results on the timestamp protocol are somewhat fortified by the results obtained independently by Agrawal and DeWitt [AgDe85], who show that optimistic protocols that use aborts instead of waits for resolving conflicts give rise to a large number of aborts that limit their throughput. However, it is generally regarded that the timestamp protocol is quite useful for concurrency control in distributed database systems because it does not have the need for orphan detectors (for orphaned locks), global deadlock detectors which add to the processing and message overheads.

Also, our results are relevant in the centralized case. Though, we believe, that the performance of the concurrency control protocols in centralized and distributed environments should follow similar patters, this can only be shown by a distributed simulation.

As mentioned earlier, we were surprised that *5C* performance was poorer (though insignificantly) than the *2PL* performance when the loads were low. We had expected that the *5C* performance would be identical to *2PL* when there were nearly no lock conflicts. The reason for this anomaly turns out not to be locking delays, but the *"bursty"* nature of transactions running under *5C*. All transaction perform read I/O at the onset, consume CPU cycles only while running, and finally generate another burst of I/O while committing when run under the *5C* protocol. This causes underutilization of resources as either the CPU or the I/O devices are forced to idle. With a small number of transactions, the transactions get into partial synchrony, that is, most of them are doing I/O, or consuming CPU cycles. We noted this phenomenon by looking at snapshots of the CPU and I/O queues. This somewhat poorer utilization of CPU and I/O devices causes the throughput to suffer discernably.

The Five Color (*FC*) protocol was significantly better than the *2PL* protocol in overall tests. The protocol performed very well for all kinds of load and conflict and did not exhibit any thrashing or large number of deadlocks. The protocol has been designed to ensure relative deadlock freedom and to allow larger throughput by through early release of locks (and use a non-2-phase locking pattern). The protocol lived up to our expectations in the simulation tests. We feel, considering that the *FC* protocol was quite close to NO in all conditions, that is is probably as close to the elusive *ideal* concurrency control protocol as we can hope to get.

We also conclude tha that the *2PL* protocol is significantly better that what most researchers give it credit for. In fact, given the small size of the window between *2PL* and *NO* in the medium conflict situations and the simplicity and ease of implementation of *2PL*, not choosing *2PL* for those database applications that do not envision high conflict rates, may be hard to justify.

8. References

[AgCaLi85] Agrawal R., Carey M. J., and Linvy M. *Models for Studying Concurrency Control Performance: Alternatives and Implications.* Proceedings of the ACM SIGMOD International Conference on Management of Data, May 1985.

[AgDe85] Agrawal R. and DeWitt D. J., *Integrated Concurrency Control and Recovery Mechanism: Design and Performance Evaluation* ACM Transactions on Database Systems, December 1985.

[Ah83] Ahuja M., *A comparison of performance of two phase locking protocols with or without planned scheduling*, Dept. TR., Dept. of Comp. Science., UT at Austin, August 1983.

[AhBrSi84] Ahuja M. L., Browne J. C. and Silberschatz A. *Optimal Throughput Scheduling for Distributed Concurrent Execution in Data Base Systems* Proc. 1984 International Conference on Parallel Processing. 251-254.

[BeGo80A] Bernstein, P.A. and Goodman, N. *Fundamental Algorithms for Concurrency Control in Distributed Systems,* CCA Tech Report, CCA-80-05.

[BeGo80B] Bernstein, P.A. and Goodman, N. *Timestamp Based Algorithms for Concurrency Control in Distributed Database Systems,* Proc. 6th International Conf. on Very Large Data Bases, Oct. 1980.

[BeGo82A] Bernstein, P.A. and Goodman, N. *Concurrency Control Algorithms for Multiversion Database Systems.* Proc. ACM SIGACT/SIGOPS Symp. on Principles of Distributed Computing, (1982)

[BeGo82B] Bernstein A.P., Goodman N., *A Sophisticates Introduction To Distritubed Database Concurrency Control*, TR-19-32, Harvard University.

[BeShRo80] Bernstein, P.A., Shipman, D.W., and Rothnie Jr., J.B. *Concurrency Control in a System for Distributed Databases (SDD-1).* ACM Trans. on Database Systems 5:1, pp. 18-51 (1980).

[Ca83] Carey M., *Modeling and Evaluation of Database Concurrency Control Algorithms*, Ph.D. dissertation, University of California, Berkeley, Sept. 83.

[CaSt84] Carey M. and Stonebraker M., *The performance of Concurrency Control Algorithms for DBMs* Proceedings of the 10th International Conference on Very Large Data Bases, Aug. 84.

[Da83] Date, C.J. *An Introduction to Database Systems*, Vol 2., Addison-Wesley (1983).

[DaKe83] Dasgupta P. and Kedem, Z.M. *A Non-2-Phase Locking Protocol for General Databases*. 8th International Conf. on Very Large Data Bases, Oct 1983.

[Da84] Dasgupta P., *Database Concurrency Control: Versatile Approaches to Improve Performance*, Ph.D. dissertation, State University of New York, Stony Brook, Sept. 1984.

[DaKe86] Dasgupta P. and Kedem Z.M., *The Five Color Concurrency Control Protocol: Non-Two-Phase Locking in General Databases.*, submitted for publication.

[EsGrLoTr76]
Eswaran, K.E., Gray, J.N., Lorie R.A., and Traiger, I.L. *On notions of Consistency and Predicate Locks in a Database System*, Comm. ACM 14:11, pp. 624-634 (1976).

[Ga78] Garcia-Molina, Hector, *A performance Comparison of two undate Algorithms for Distributed Data management and Computer Networks, p. 108-122.*

[GoSu83] Goodman N., Suri R., *A simple analytical model of exclusive locking in Data Base Systems*, Tr-01-83 Harvard University, Center for Research in Computing Technology.

[GeSe78] Gelenbe E., and Sevcik K., *Analysis of update synchronization for multiple copy data base*, ACM TODS, Vol. 7, No. 2, June 1982.

[Gr78] Gray J.N. *Notes on Data Base Operating Systems*. IBM Research Report RJ2188, Feb 1978.

[GrHoOb81]Gray J., Pete Homan, Obermark R., Korth H., *A straw man analysis of probability of waiting and deadlock*, IBM research report FJ3066 (38112), San Jose, CA. 1981.

[KiLa83] Kiessling W., Landherr G. *A Quantatitive Comparison of Lockprotocols for Centralized Databases*. Proc. 9th International Conf. on Very Large Data Bases, Oct. 83.

[KrDa82] Krishnamurthy R., Dayal U., *Theory of Serializability for a Parallel Model of Transactions*, ACM 1982, pp 293 - 305.

[KuPa79] Kung H.T., Papadimitriou C.H., *An Optimality Theory of Database Concurrency Control*, Proc. ACM SIGMOD conference, 1979.

[LiNo82a] Lin W.K., Nolte J., *Performance of two-phase locking* 6th Berkeley Workshop on Distributed Computing Systems, 1982.

[Li81] Lin, W.K. *Performance Evaluation of Two Concurrency Control Mechanisms in a Distributed Database System*, Proc. 1981 ACM SIGMOD Conf., p. 84-92 (1981).

[LiNo83] Lin, W.K., Nolte J., *Basic Timestamp, Multiple Version Timestamp, and Two-Phase Locking*, 9th Conference on VLDB, 1983.

[Li79] Lindsay B.G. et al., *Notes on Distributed Database Systems*. IBM Research Report RJ2571(33471) 7/145/79.

[MuKr77] Muntz, R. and Krenz, G. *Concurrency in Database Systems - A Simulation Study*. Proc. ACM SIGMOD Conf., (1977).

[Pa79] Papadimitriou, C.H. *The Serializability of Concurrent Database Updates*, J. ACM 26:4, pp. 631-653 (1979).

[Pa81] Papadimitriou C.H., *On the power of locking*, Proc. of 1981 ACM SIGMOD conference.

[Re79] Reis, D.R., *The Effects of Concurrency Control on the Performance of Distributed Data Management Systems*, Proc. 4th Berkeley Workshop on Dist. Data Management and Computer Networks, p.75-112 (1979)

[RoStLe78] Rosenkrantz, D.J., Stearns, R.I, and Lewis II, P.M. *System Level Concurrency Control for Distributed Database Systems.* ACM Transactions on Database Systems, 3:2 pp. 178-198 (1978).

[Se81] Sevcik K.C., *Database system performance prediction using an analytical model*, 7th Conference on VLDB, Cannes, France, 1981.

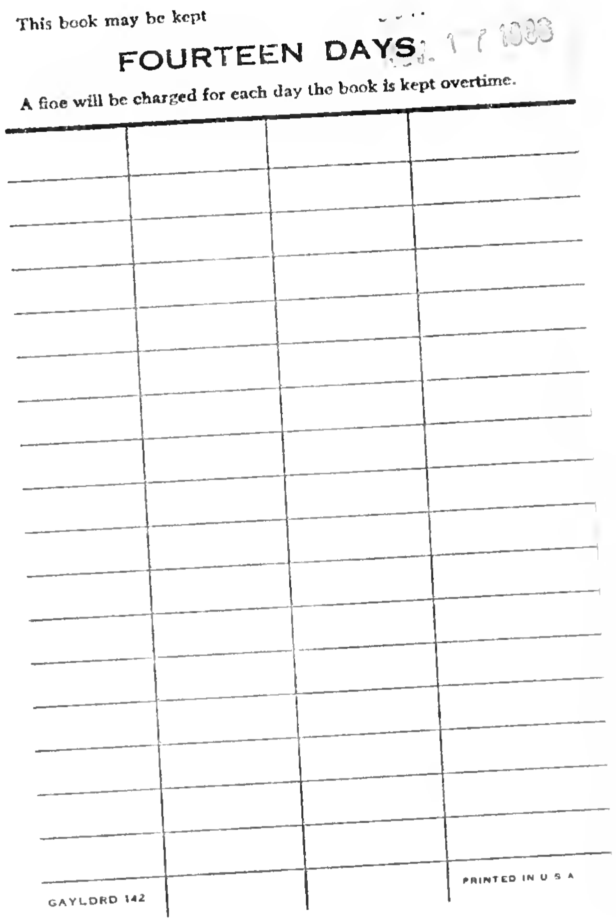
[StLeRo76] Stearns, R.E., Lewis II, P.M., and Rosenkrantz, D.J. *Concurrency Control for Database Systems.* Proc. 17th IEEE FOCS, pp. 19-32 (1976).

[Ta84] Tay Y. C. *A Mean Value Performance Model for Locking in Databases* Ph.D. dissertation, Harvard University, Feb. 1984.

[TaGoSu85] Tay, Y. C., Goodman N. and Suri R., *Locking Performance in Centralized Databases* ACM Transaction on Database Systems, December 1985.

[Th85]
Thomasian, A. *Performance evaluation for Centralized Databases with Static Locking* IEEE Transactions on Software Engineering, April 1985.

[Ul82] Ullman J.D. *Principles of Database Systems.*, 2nd ed., Computer Science Press (1982), Potomac MD.



NYU COMPSCI TR-214
Dasgupta, Partha
Simulating distributed
database operating systems
and evaluating... c.2